OCaml étendu avec du filtrage par comotifs

Paul Laforgue, Yann Régis-Gianas

OCaml étendu avec du filtrage par comotifs

Paul Laforgue[1] and Yann Régis-Gianas[2]

[1] Univ Paris Diderot, Paris, France
paul.laforgue123@gmail.com
[2] Univ Paris Diderot, Sorbonne Paris Cité, IRIF/PPS, UMR 8243 CNRS, PiR2, INRIA
Paris-Rocquencourt, Paris, France
yrg@irif.fr

Résumé

Cet article décrit une extension du langage de programmation OCaml avec de nouvelles constructions pour définir et manipuler des structures de données infinies: les types de données coalgébriques et le filtrage par comotifs.

1 Introduction

Les types de données algébriques et le filtrage par motifs sont des mécanismes d'OCaml particulièrement utiles pour spécifier et définir des programmes travaillant sur des objets finis et inductifs, comme les listes et les arbres par exemple. Le programmeur OCaml apprécie la saveur mathématique et la simplicité calculatoire de ces mécanismes car ce sont les ingrédients d'une programmation sûre et efficace.

Concernant le traitement des objets infinis, le tableau est moins reluisant. Prenons le cas des listes infinies et essayons de définir la liste des entiers naturels:

```
type nat = Zero | Succ of nat
let rec from : nat → nat list = fun n → n :: from (Succ n)
let naturals = from Zero
let rec nth n s = match n with
  | Zero → List.hd s
  | Succ m → nth m (List.tl s)
```

Comme OCaml est un langage strict, l'évaluation de naturals provoque une divergence car la construction des entiers naturels s'opère d'un coup. Au contraire, on aimerait que la construction des objets infinis ne se fasse qu'à la demande et partiellement, lorsque l'on doit en *observer* un fragment dont dépend un calcul.

Fort de cette idée, nous pouvons utiliser le mot-clé **lazy** pour retarder systématiquement la construction des listes infinies, et invoquer Lazy.force pour calculer une à une les têtes de ces listes au moment où le calcul le nécessite:

```
type 'a stream = ('a cell) Lazy.t and 'a cell = Cell of 'a * 'a stream
let rec from : nat → nat stream = fun n → lazy (Cell (n, from (Succ n)))
let naturals = from Zero
let rec nth : nat → 'a stream → 'a = fun n s →
  let Cell (hd, tl) = Lazy.force s in
  match n with
  | Zero → hd
  | Succ m → nth m tl
```

Si les types et les opérations du module Lazy rendent possible le calcul sur des structures infinies, elles obscurcissent aussi la définition de naturals en introduisant des considérations de bas-niveau sur l'ordre des calculs, absentes des définitions mathématiques habituelles.

Le filtrage par comotifs[1] est une généralisation du filtrage par motifs. En OCaml avec comotifs, un objet infini est défini sans précaution particulière vis-à-vis de la divergence:

```
type 'a stream = {Head : 'a; Tail : 'a stream}
let corec from : nat → nat stream with
  | (.. n)#Head → n
  | (.. n)#Tail → from (Succ n)
let naturals = from Zero
let rec nth : nat → 'a stream → 'a = fun n s → match n with
  | Zero → s#Head
  | Succ m → nth m s#Tail
```

Contrairement aux types algébriques qui sont définis par des constructeurs, les types coalgébriques sont définis par des destructeurs, aussi appelés observations. Défini ici en ligne 1, le type coalgébrique 'a stream a deux destructeurs: Head produisant une valeur de type 'a et Tail produisant une valeur de type 'a stream.

En ligne 2, from est défini par filtrage de comotifs. Pour cela, le programmeur raisonne par cas sur les différentes observations, Head et Tail, qui peuvent s'appliquer à from n. Dans un comotif, la notation .. réfère à l'objet infini observé et #O à l'observation O que l'on est en train de définir dans la branche courante. Ainsi, la ligne 3 se lit: "Si on observe la tête de from n alors renvoyer n." Tandis que la ligne 4 se lit: "Si on observe la suite de from n alors calculer from (Succ n)". Remarquez que contrairement à un filtrage par motifs classique, les deux branches n'ont pas le même type: la première a le type 'a et la seconde le type 'a stream.

Ce sont donc les applications d'observations qui déclenchent les calculs des structures infinies. De telles applications se retrouvent dans la définition de nth: s#Head déclenche le calcul de la tête de s, s#Tail celui de la suite de s.

Nous avons précédemment montré[2] que l'on peut étendre OCaml avec des comotifs à l'aide d'une simple macro: une transformation syntaxique purement locale qui a lieu entre l'analyse syntaxique et le typeur. Notre prototype est accessible en utilisant le gestionnaire de paquet `opam` et la commande: `opam switch 4.04+copatterns`.

Plan La section 2 décrit informellement notre transformation à travers un exemple. Le filtrage par comotifs est un mécanisme de haut niveau qui supporte des constructions avancées allant au-delà de notre exemple introductif (imbrication, inclusion des motifs, prise en compte d'égalités de type). Elles se seront présentées en section 3. Une particularité de notre encodage est d'introduire les observations comme des objets de première classe et de premier ordre. Cette caractéristique offre un nouveau champ d'action au programmeur et nous l'illustrons en section 4. Finalement, nous concluons et parlons de travaux futurs en section 5.

2 Transformation

Dès qu'un langage de programmation fonctionnelle est muni de types algébriques généralisés (GADTs) et de polymorphisme de second-ordre, on peut étendre son analyse par motifs en analyse par comotifs à l'aide d'une transformation syntaxique purement locale[2]. En un mot, cette transformation traduit toute valeur coalgébrique définie par une analyse par comotifs en une fonction définie par cas sur la forme des observations de ce type coalgébrique. Dans cette section, nous présentons ce résultat informellement en décrivant les images par notre transformation de la définition du type stream et de l'analyse par comotifs définissant from.

La traduction de la déclaration du type stream commence par introduire un nouveau type stream_obs correspondant aux observations du type stream:

```
type ('a, 'o) stream_obs =
  | Head : ('a, 'a) stream_obs
  | Tail :  ('a, 'a stream) stream_obs
```

Ce type possède deux constructeurs: Head et Tail. Le premier paramètre de type 'a correspond tout simplement à celui de stream, il s'agit du type des éléments de la liste infinie. Le second paramètre 'o est plus intéressant: il correspond au type de retour des observations. Notons que les deux constructeurs instancient différemment 'o : pour Head, 'o vaut 'a alors que pour Tail, 'o vaut 'a stream. Cette spécialisation des schémas de type de chaque constructeur est caractéristique d'un GADT.

Le type coalgébrique stream est traduit par un type algébrique à un seul constructeur nommé Stream par convention. Celui-ci prend en argument une fonction qui attend une observation en entrée et qui est polymorphe vis-à-vis du type de retour de cette observation. Comme il est d'usage en OCaml, ce polymorphisme de second-ordre est introduit grâce à un enregistrement dont l'unique champ dispatch a un type polymorphe:

```
type 'a stream = Stream of {dispatch : 'o.('a, 'o) stream_obs → 'o}
```

Ici la fonction dispatch a donc pour type $\forall\sigma.(\alpha,\sigma)$ stream_obs $\to \sigma$. Son type, qui n'est pas sans rappeler celui d'un terme en forme CPS, est le témoin d'une inversion de contrôle entre l'environnement d'évaluation et la valeur coalgébrique.

Naturellement, la transformation du filtrage par comotifs est dirigée par celle des types coalgébriques. Lors de la transformation d'un filtrage par comotifs, une fonction auxiliaire dispatch est systématiquement créée. Dans le cas très simple de notre exemple, cette fonction est une analyse par motifs qui a exactement la même structure que l'analyse par comotifs qu'elle représente. Pour les analyses par comotifs plus complexes de la section 3, l'idée fondamentale reste la même: en effet, les analyses complexes peuvent s'exprimer par imbrication d'analyses par comotifs dites *simples*[2].

```
let rec from : nat → nat stream = fun n →
  let dispatch : type o.(o, nat) stream_obs → o = function
    | Head → n
    | Tail → from (Succ n)
  in Stream {dispatch}
```

Notez que cette fonction est bien typée car (i) dans la première branche o et nat sont équivalents donc n de type nat est bien du type attendu o ; (ii) dans la seconde branche o et nat stream sont équivalents, donc from (Succ n) de type nat stream est bien du type attendu o.

Dernier ingrédient de notre traduction: le cas de l'application des observations. Il s'agit tout simplement d'appliquer la fonction dispatch sur l'observation considérée. On se dote de:

```
let head {dispatch} = dispatch Head
let tail {dispatch} = dispatch Tail
```

ce qui permet de traduire l'observation s#Head en l'application head s.

3 Constructions avancées du filtrage par comotifs

Observations imbriquées On peut chaîner les observations au sein d'une même analyse[1]:

```
let corec fib : int stream with
  | ..#Head → 0
  | ..#Tail : int stream with
  | ..#Tail#Head → 1
  | ..#Tail#Tail → map2 (+) fib fib#Tail
```

En ligne 3, la syntaxe ..#Tail : int stream **with** introduit une sous-analyse par comotifs du stream produit par l'observation Tail de fib. Notez l'annotation de type qui doit obligatoirement préciser le type de ..#Tail: c'est une limite de notre approche dont nous reparlons en section 5.

Mémoïzation dans le filtrage Dans l'exemple précédent, les deux occurrences de fib dans l'appel à map2 produisent deux flux distincts en mémoire ne partageant aucun de leurs calculs: par conséquent, la complexité du calcul d'un élément de rang n de cette suite est exponentielle en n. En préfixant du mot-clé **lazy** la définition de fib, notre transformation introduit une mémoïzation qui garantit que les observations de fib (et de ses sous-flux) sont partagées, ce qui restaure la complexité linéaire de cet algorithme:

```
let corec lazy fib : int stream with
  | ..#Head → 0
  | ..#Tail : int stream with
  | ...#Head → 1
  | ...#Tail → map2 (+) fib fib#Tail
```

Des motifs dans les comotifs Un comotif peut utiliser un motif pour filtrer sur certains sous-ensembles des arguments d'un objet infini. Par exemple, la liste infinie cycle n suivante est de la forme $[\mathsf{n}, \mathsf{n}-1, \ldots, 1, 0, 3, 2, 1, 0, 3, \ldots]$:

```
let corec cycle : nat → nat stream with
  | (.. n)#Head → n
  | (.. Zero)#Tail → cycle (Succ (Succ (Succ Zero)))
  | (.. (Succ n))#Tail → cycle n
```

cycle est un flux paramétré par un entier n. En ligne 2, l'argument n est la réponse à l'observation de la tête de cycle n. L'observation de la suite dépend de la forme de n: si n est Zero, la séquence reprend à Succ (Succ (Succ Zero)), sinon le cycle se poursuit au prédecesseur de n.

Types coalgébriques indexés Comme les GADTs, les types coalgébriques peuvent être indexés de façon à capturer des invariants statiques permettant de justifier l'absence de certains cas absurdes dans les analyses par comotifs. Par exemple, un objet coalgébrique de type ('a, 'b) bounded_iterator encapsule une collection d'élements de type 'a et restreint statiquement la quantité d'éléments de cette collection auquelle le client a accès. On se donne les types suivants:

```
type 'a fuel = Dry : zero fuel | More : 'a fuel → ('a succ) fuel
type ('a, β) bounded_iterator = {
  GetVal : 'a;
  Next : ('a, β) bounded_iterator ← ('a, β succ) bounded_iterator;
}
```

[1] La définition de map2 est laissée en exercice au lecteur.

Le GADT 'a fuel représente la notion de crédits donnés au client de l'itérateur[2]. Le premier paramètre du type bounded_iterator représente le type des élements de la collection et le second paramètre le nombre de crédits encore disponibles pour le client de l'itérateur. La structure héberge une valeur de type 'a observable par GetVal. L'observation Next renvoie un nouvel objet dont le nombre de crédits a été décrémenté. Le type de cette observation est indexé par β succ. On contraint ainsi l'observation Next à être uniquement appliquée aux itérateurs qui disposent d'un nombre strictement positif de crédits. Tout module qui offre un type de collection 'a t muni de deux opérations head et tail peut désormais être étendu par une notion d'itérateur borné à l'aide du foncteur MkIterator suivant:

```
module type Seq = sig
  type 'a t
  val head : 'a t → 'a
  val tl : 'a t → 'a t
end
```

```
module MkIterator (S : Seq) = struct
  let corec wrap : type a b.a S.t → b fuel → (a, b) iterator with
      | (.. l n)#GetVal → S.head l
      | (.. l (More n))#Next → wrap (S.tl l) n
end
```

L'analyse par comotifs de la fonction wrap n'a pas de code pour le cas où le fuel vaut Dry et l'observation est Next. C'est logique! Un tel cas serait mal typé puisqu'il imposerait au paramètre b d'être à la fois de la forme zero et de la forme β succ, deux types incompatibles.

4 Destructeurs de premier ordre et première classe

En section 2, nous avons montré que pour chaque type coalgébrique notre transformation introduit un GADT pour représenter ses observations. La définition de ce GADT suit une convention de nommage documentée ce qui permet au programmeur de faire explicitement référence aux constructeurs de données et de type de ce GADT. De cette façon, les destructeurs peuvent être utilisés dans le calcul comme n'importe quelle autre valeur. Comme ils sont représentés par de simples constructeurs, ils peuvent aussi être comparés ou sérialisés.

Pour illustrer ce mécanisme, considérons le scénario suivant. Le programmeur définit un type enregistrement loc avec trois champs (name de type string, une abscisse x et une ordonnée y de type int) et souhaite offrir des fonctions de sélection et de mise-à-jour de ces champs. Pour cela, il est confronté à la pénible tâche de devoir répéter du code:

```
type loc = {name : string; x : int; y : int}
let select_name lc = lc.name     and update_name s lc = {lc with name = s}
let select_x    lc = lc.x        and update_x    b lc = {lc with x = b}
let select_y    lc = lc.y        and update_y    n lc = {lc with y = n}
```

Ce problème est dû au fait que les étiquettes ne sont pas des objets de première classe en OCaml. Nous aimerions disposer de combinateurs génériques tels que select ou update auxquels nous n'aurions qu'à passer l'étiquette concernée en argument. Reprenons donc le même scénario mais cette fois-ci, au lieu d'un type enregistrement, nous utilisons un type coalgébrique.

```
type loc = {Name : string; X : int; Y : int}
```

Le combinateur select peut alors s'exprimer très naturellement. Il suffit d'extraire la fonction dispatch d'une valeur de type loc, puis de l'appliquer à un destructeur d passé en argument.

```
let select (d : 'a loc_obs) (Loc {dispatch} : loc) : 'a = dispatch d
```

[2] On suppose que les constructeurs de types zero et succ sont distincts.

Définir update requiert davantage de travail. En première approximation, on pourrait écrire:

```
let update (type a) (d1 : a loc_obs) (x : a) (Loc {dispatch}) =
  let dispatch : type o.o loc_obs → o = fun d2 →
    if d1 = d2 then x else dispatch d2
  in Loc {dispatch}
```

mais ce programme est mal typé! D'une part, d1 et d2 n'ont pas nécessairement le même type et d'autre part, rien n'informe ici le typeur que x a le type o. Fort heureusement, on peut facilement écrire une fonction de comparaison eq_loc[3] dont le type de retour est plus riche qu'un simple booléen: dans le cas positif, cette fonction retourne une preuve d'égalité entre les indices des types des deux constructeurs:

```
type (_, _) eq = Eq : ('a,' a) eq
val eq_loc : type a b.a loc_obs * b loc_obs → ((a, b) eq) option
```

Dès lors, la fonction update peut être corrigée en utilisant eq_loc:

```
let update (type a) (d1 : a loc_obs) (x : a) (Loc {dispatch}) =
  let dispatch : type o.o loc_obs → o = fun d2 → match eq_loc (d1, d2) with
    | Some Eq → x
    | _ → dispatch d2
  in Loc {dispatch}
```

Cette fonction est bien typée[4] car eq_loc accepte des arguments dont les indices diffèrent et dans le cas Some Eq, l'expression x est typée dans un contexte de typage enrichi par l'égalité a = o qui permet de lui affecter le type o.

5 Conclusion et perspectives

Pour résumer, OCaml peut être étendu par des comotifs sans trop d'effort et la syntaxe des comotifs permet de définir des objets infinis de façon concise, richement typée, et efficace.

Notre approche purement syntaxique impose néanmoins au programmeur d'annoter par leur type toutes les analyses par comotifs. Ceci nous permet de construire une annotation de type pour la fonction dispatch introduite par la transformation et ainsi de nous assurer que le raffinement de type de l'analyse par motifs de dispatch a bien lieu. En déplaçant notre transformation à l'intérieur du moteur de typage d'OCaml, une partie de ces annotations pourraient être automatiquement inférées.

References

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. In *ACM SIGPLAN Notices*, volume 48, pages 27–38. ACM, 2013.

[2] Paul Laforgue and Yann Régis-Gianas. Copattern matching and first-class observations in ocaml, with a macro. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, PPDP '17, pages 97–108, New York, NY, USA, 2017. ACM.

[3] Laissé en exercice au lecteur.

[4] Mais notons qu'elle est très inefficace! Une implémentation plus réaliste pourrait peut-être s'appuyer sur les dictionnaires à clés hétérogènes fournis par le module hmap de Daniel C. Bünzli.